Hardware-Accelerated Range Partitioning

Lisa Wu, Raymond J. Barker, Martha A. Kim, Kenneth A. Ross
Department of Computer Science, Columbia University, New York, New York, USA
{lisa,rjb2150,martha,kar}@cs.columbia.edu

Abstract

With global pool of data growing at over 2.5 quinitillion bytes per day and over 90% of all data in existence created in the last two years alone [23], there can be little doubt that we have entered the big data era. This trend has brought database performance to the forefront of high throughput, low energy system design. This paper explores targeted deployment of hardware accelerators to improve the throughput and efficiency of database processing. Partitioning, a critical operation when manipulating large data sets, is often the limiting factor in database performance, and represents a significant amount of the overall runtime of database processing workloads.

This paper describes a hardware-software streaming framework and a hardware accelerator for range partitioning, or HARP. The streaming framework offers seamless execution environment for database processing elements such as HARP. HARP offers performance, as well as orders of magnitude gains in power and area efficiency. A detailed analysis of a 32nm physical design shows 9.3 times the throughput of a highly optimized and optimistic software implementation, while consuming just 3.6% of the area and 2.6% of the power of a single Xeon core in the same technology generation.

1. Introduction

In the era of big data, fields as varied as natural language processing, medical science, national security, and business management all rely on the ability to sift through and analyze massive, multi-dimensional data sets. These communities rely on computer systems to quickly and efficiently process vast volumes of data. In this work we explore deploying specialized hardware to more effectively address this task.

Databases are designed to help managing large quantities of data, allowing users to query and update the information they contain. The database community has been developing algorithms to support fast or even real-time queries over relational databases. As data sizes grow, big databases increasingly *partition* the data among multiple tasks that operate on sub tables instead of the large database. Partitioning, as illustrated in the small example in Figure 1, assigns each record in a large table to a smaller table based on the value of a particular field in the record, which is the transaction date in Figure 1. Partitioning enables the resulting partitions to be processed independently and more efficiently (e.g., in parallel and with better cache locality). Partitioning is used in virtually all modern database systems including Oracle Database 11g [39], IBM DB2 [22],

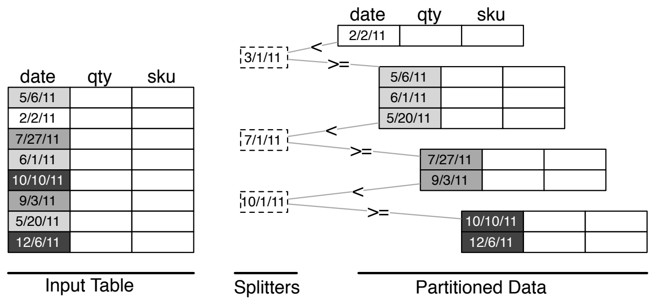

Figure 1: An example table of sales records *range partitioned* by date, into smaller tables. Processing big data one partition at a time makes working sets cache-resident, dramatically improving the overall analysis speed.

and Microsoft SQL Server 2012 [33] to improve performance, manageability and availability in the face of big data, and the partitioning step itself has become a key determinant of query processing in big database.

Now is the time to explore custom hardware for handling large data. At the same moment we hit the power wall, performance gains of commodity hardware started slowing, and data sizes started to explode. As the price of memory drops, modern databases are not typically disk I/O bound [1, 19], with many databases now either fitting into main memory or having a memory-resident working set.

We present a design for a an ASIC Hardware-Accelerated Range Partitioner, or HARP. We demonstrate that software implementations of partitioning on a general purpose processor have fundamental performance limitations that cause partitioning to be a bottleneck. HARP both accelerates the data partitioning itself and frees up processors for other computations. Since database and other data processing systems represent a common high-value server workload in this age of big data, the impact of improvements in partitioning performance would be widespread.

This paper makes the following four contributions.

- A detailed software scaling analysis range partitioning. Our analysis quantifies how severely the severity of the bottleneck, even after existing techniques such as SIMD instructions and multi-threading are deployed (Section 3).
- A hardware-software architecture for data stream processing. Data streams are managed via explicit software instructions and hardware stream buffers from which they are processed by HARP, a specialized processing element for range partitioning (Section 4).

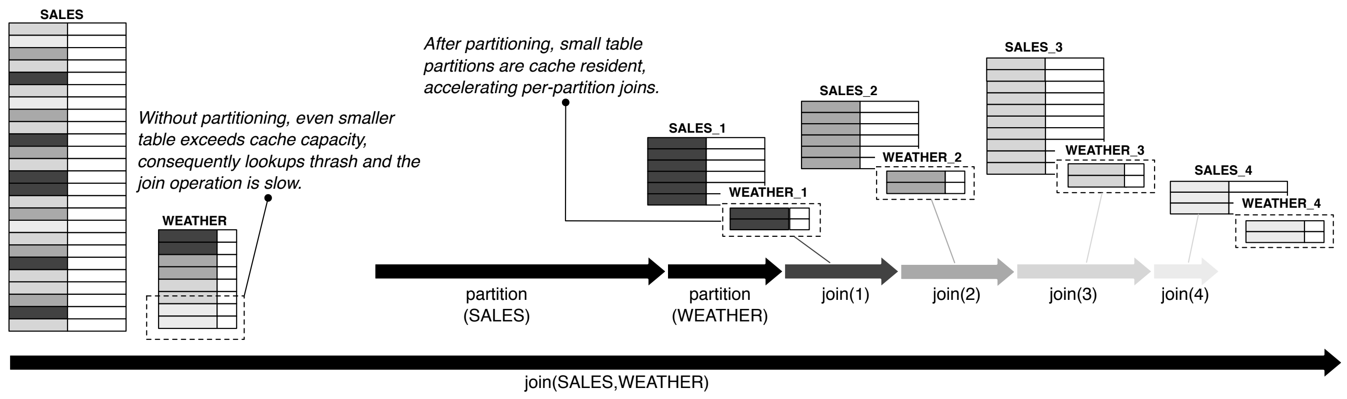
SALES
WEATHER
Without partitioning, even smaller table exceeds cache capacity, consequently lookups thrash and the join operation is slow.
After partitioning, small table partitions are cache resident, accelerating per-partition joins.
SALES_1
WEATHER_1
SALES_2
WEATHER_2
SALES_3
WEATHER_3
SALES_4
WEATHER_4
partition (SALES)
partition (WEATHER)
join(1)
join(2)
join(3)
join(4)
join(SALES,WEATHER)

Figure 2: Joining two large tables easily exceeds cache capacity. State of the art join implementations, partition the tables to be joined then compute partition-wise joins, each of which exhibits substantially improved cache locality [28, 2]. Joins are extremely expensive on large datasets, and partitioning represents a significant fraction, up to half, of the observed join time [28].

- An evaluation of the area, power, and efficiency of the HARP design. Synthesized, placed and routed, a single HARP unit would occupy just 3.3% of the area of a commodity Xeon processor core and can process up to 3.72 GB/sec of input (at least 9.3 times more than a single Xeon core), matching the throughput of 16 software threads (Section 5.2).
- A detailed sensitivity analysis of HARP designs. These studies quantify tradeoffs between specialization and flexibility. We demonstrate that HARP tolerates unbalanced data distribution, that throughput is most sensitive to record width, and that area and power are directly proportional to the partitioning fanout (Section 5.3-Section 5.6).

2. Background and Motivation

Here we provide some background on the range partitioning operation and its deployment and prevalence in databases.

Range Partitioning. Partitioning a table splits the table into multiple smaller tables called *partitions*. Every row in the original table will be assigned to one partition, based on its value in the *key* field. Figure 1 shows an example table of sales transactions partitioned based on the date of the transaction. For each record in the table the destination partition is computed from the key value. This work focuses on a method called *range partitioning* which splits the space of keys into contiguous ranges, for instance by quarter as illustrated in Figure 1. The boundary values of these ranges are called *splitters*.

Benefits of Partitioning. Partitioning large data tables has many benefits. Once tables become so large that the tables themselves or their associated processing metadata cannot fit into a cache, the analysis speed drops off rapidly. The partitioned tables can be processed more expeditiously. When a quarterly sales report is needed, instead of having to process all the sales records, only the partitioned tables containing desired quarter(s) need to be processed. Another use case is to provide real-time sales updates. Since the records are partitioned by date, if a new sale is made, the only partition that needs updating is the partition that contains today's date. You can also imagine distributing various partitions of a huge database to different locations. If the sales records were to be partitioned by geographical regions, then the sale records of New York can be stored in a facility on the east coast for faster access.

Partition-Wise Joins. A commonly used database operation is a *join* operation. A join takes a common key from two different tables and creates a new table containing information from both tables based on a particular condition. For example, if one wants to analyze whether the weather effects sales, one would *join* the sales records with the weather records on the date of each transaction. In this case records from the `SALES` table are joined with records from the `WEATHER` table where `SALES.date == WEATHER.date`. If the `WEATHER` table is too large to fit in the cache, this whole process will have very poor cache locality, as illustrated in Figure 2-left. On the other hand, if both tables are partitioned on date, each partition can be joined in a pairwise fashion illustrated by Figure 2-right. When each partition of the `WEATHER` table fits in the cache, joining each partition can proceed much more rapidly. When the data is large enough, the time spent partitioning is more than offset by the time saved with the resulting cache-friendly partition-wise joins.

Most queries begin with one ore more joins to cross reference tables. Already join operations are critical to database performance, as evidenced by the substantial optimization and many different implementations tailored to an array of possible scenarios. Because query optimizers strive to filter the data as soon as possible after the join, the initial join operation is expected to further dominate as data sizes grow. Analyses of the state of the art in join implementations have indicate that up to half of the time spent performing a join is attributable to range partitioning [28].

Partitioning Desirata. A good partitioning algorithm will have several properties. One useful property of range partitioning is that the resulting partitions are ordered. **Ordered partitions** are useful when a global sort of the data is required. A second property is **record order preservation**, specifically that all records in a partition appear in the same order they were found in the input record stream. This property is important for some algorithms, including radix sorting. While range partitioning itself does not enforce this, our hardware implementation HARP does have this property.

A third property is **skew-tolerance**. *Skew* is the term for uneven data distribution across partitions. In the sales records example above, if summer sales far exceed winter sales, then the date ranges in the summer months would *skew* the partitioning operation. Skew result in one or more partitions being much larger than others and negating many of the benefits of partitioning. HARP handles skew at two levels. First, we implement a variant of range partitioning described by Ross and Cieslewicz [43] that is less vulnerable to skew. This scheme includes one *equality partition* per splitter, where the entire partition gets just a single key. If the user assigns expected hot keys to be splitters, they will get their own partition, helping to balance the data across partitions. Second, the HARP microarchitecture is skew tolerant meaning that in the presence of skew, the partitioning throughput degrades minimally.

3. Software Range Partitioning Analysis

Here we identify performance limitations for software partitioning on general purpose CPUs. In Section 3.2 we analyze single-core performance in detail. Since partitioning performance scales with additional cores are used [8, 28, 2], in Section 3.3 we quantify multithreaded performance as well.

3.1. Methodology

We characterize an optimistic partitioning microbenchmark whose inner loop runs over an input table reading in a record at a time, r, computing r's partition using a partition function $p = partitionfunction(r)$, and then writing r to the destination partition p.

We experiment with 8 byte records (4 byte key, 4 byte payload) as in [28] and 16 byte records (8 byte key and 8 byte payload) as in [8, 2]. The latter configuration will be most directly comparable to our hardware partitioner that is tuned for 16 byte records. Keys and payloads are stored columnwise in separate arrays.[1]

We evaluate different partitioning methods by varying the (inlined) implementation of $partitionfunction()$. We considered range partitioning and three other methods for comparison:

[1]When the key and payload were stored contiguously, we saw an improvement in performance near the TLB capacity, because the TLB footprint is half as big.

1. No partitioning: All items go to a single partition. This method is similar in behavior to `memcpy` and provides a baseline software throughput.
2. Hash partitioning: Partitions are computed using a multiplicative hash of each record's the key value.
3. Direct partitioning: Like hash partitioning, but avoids hashing by treating the key itself as the hash value.
4. Range partitioning: Equality range partitioning, using the state of the art algorithm from [43]. This implementation requires a binary search of the splitter array.

Our software measurements are optimistic. We use the ideal input distributinon (i.e., uniform random), and in the case of range partitioning, splitters were also uniformly distributed. Real-world data sets will typically have less favorable characteristics. The measurements are also optimistic with respect to output data layout. Prior to partitioning it is impossible to know exactly how many records will land in each partition, making it impossible to pre-allocate the perfect amount of output memory. Kim et al. [28] make an additional pass through the input to calculate partition sizes so that partitions are free of fragmentation, arguing that since the partitioning process is compute-bound, the extra pass through the data has only a small performance impact. Another approach is simply to allocate large chunks of memory, on demand, as the partitioning operation runs. This software microbenchmark is optimistic in that it pre-allocates memory and performs no bounds checking during partitioning. Properly managing output memory can only slow performance.

The code was compiled with `gcc` version 4.4.3 at optimization level 3 and executed on an 8-core 2.4 GHz Intel Xeon E5620 [24] with 48 GB of RAM. The memory subsystem of this platform is capable of 32 GB/sec of total bandwidth. We present software performance as compute throughput in GB/sec. A partitioner operating at X GB/sec would consume 2X GB/sec of memory bandwidth (X GB/sec of data in + X GB/sec of data out). We ran each algorithm 9 times on an input of 10^8 records, and take the median measurement over the 9 runs.

3.2. Single Core Software Partitioning

Figure 3 shows the throughput of the four partitioning methods for 128-way and 256-way partitioning on 8-byte records. The range partitioning configurations used 63 and 127 splitters respectively. As a result, the number of partitions in each case are actually 127 and 255 rather than 128 and 256.

The no-partitioning method (labeled "none" in Figure 3) is similar in behavior to `memcpy`. Comparing direct and hash partitioning, the data indicate that the hash function computation incurs only mild cost. Comparing with prior results, the cycle cost of 128-way hash partitioning is close to that observed in [28] for a single core. Additionally, the significant drop in throughput between 128- and 256-way partitioning is consistent with the observations of [28] that 128-way partitioning is the largest partitioning factor that does not incur excessive L1

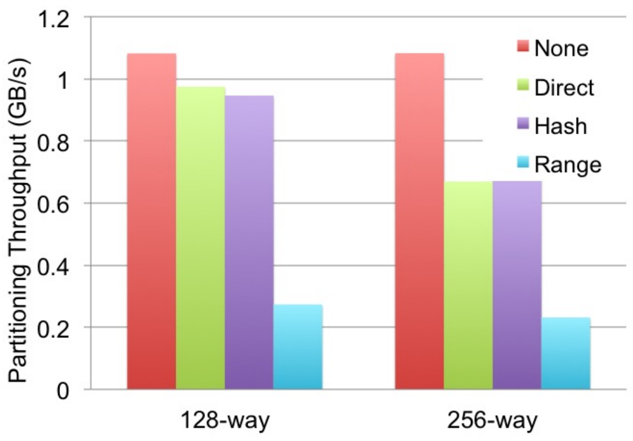

Figure 3: Compute throughput for different partitioning algorithms. Range partitioning is the most costly for both 128-way partitioning (left) and 256-way partitioning (right).

TLB thrashing.

Range-partitioning needs more cycles per record, and thus has lower throughput, because it must traverse the splitter array. It is possible to improve the splitter array lookups by using SIMD techniques such as those discussed by Schlegel et al. [45]. We downloaded the code from [45] and observed that the time for a SIMD-enhanced binary search on our machine (without actually outputting the partitions) was about 30 cycles for 128 splitters, and 50 cycles for 256 splitters. When accounting for reading and writing data, SIMD searches improve the throughput of range partitioning up to 40% (less when the L1 TLB is thrashing). Even so, this 0.38 GB/sec partitioning throughput (corresponding 0.77 GB/sec memory throughput), does not come close to using the available memory bandwidth.

3.3. Parallel Scaling

Because range partitioning scales with multiple threads, we also measured the performance of multithreaded software. Figure 4 shows the scaling behavior of 128-way hash and range partitioning for both 8 byte and 16 byte records.[2] As this data indicate, using 16 threads can improve software range partitioning throughput by a factor of ten. Ultimately, in all cases, the threads begin contending for shared resources other than memory bandwidth and the performance improvements tail off. For range partitioning, the partitioning throughput peaks with 16 threads at 2.75 GB/sec for 8 byte records and 4.5 GB/sec for 16 byte records. Given that the memory bandwidth of the system is sufficient to process 16 GB/sec of input (plus and 16 GB/sec of output for a total of 32 GB/sec), deploying all compute resources in this fashion still consumes just 17% and 28% of the available memory bandwidth.

[2]There are 16 hardware threads available, on 8 processor cores.

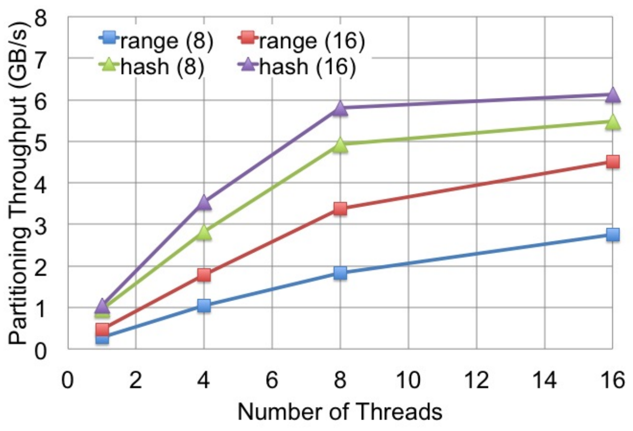

Figure 4: Partitioning bandwidth (in GB/sec of input) achieved for 128-way software partitioning using hash and range partitioning, as a function of the number of parallel threads. Data is shown for both 8 byte and 16 byte records.

4. Hardware Accelerated Range Partitioning Architecture

As the software experiments show, software is capable of saturating memory bandwidth when bringing data into a core for processing, however it becomes a bottleneck when computing the partition itself. We present a framework in which the computation is accelerated in hardware (via HARP), but all input and data stream management is left to software to maximize flexibility and simplify the interface to the accelerator. Figure 5 shows a block diagram of the major components in the system. The two stream buffers, one running from memory to HARP and the other from HARP to memory, decouple HARP from the rest of the system. One set of instruction set extensions moves data between the processor, memory, and the stream buffers. A second set of instructions provides configuration and control for the HARP accelerator, which freely pulls data from and pushes data to the stream buffers.

4.1. System Architecture

To ensure that HARP can process data at the desired throughput, the framework surrounding the accelerator must address the following three concerns: (1) records need to be readily available to HARP for partitioning, (2) records need to be written out to memory as fast as possible once HARP has processed them, and (3) the machine must continue seamless execution after an interrupt, exception, or context switch. We use a hardware-software streaming framework based on the concept outlined in Jouppi's prefetch stream buffer work [27].

Table 1 describes the instructions are used to manage the data streams. `sbload` is issued by the processor and loads data from memory into the SB_{in}. Each `sbload` takes as an argument the address from which to load memory and the number of bytes to load. `sbstore` works on reverse, taking

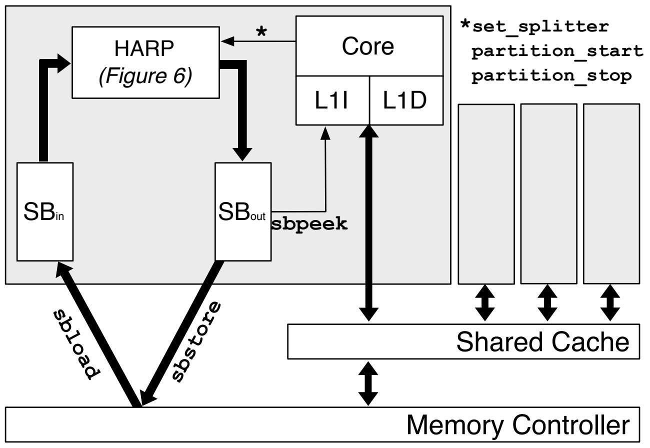

Figure 5: Block diagram of a HARP system architecture

data from the head of the SB_{out} and writing it to the specified address.

SB_{in} has a simple stride prefetcher that uses the burst size (64 bytes for our study) as the stride and prefetches up to the next 15 record bursts within the same page boundary. `sbload` checks the head of the SB_{in} FIFO before sending requests to memory as strided accesses will likely hit in the SB_{in}. In the event that the issued *sbload* misses the SB_{in} FIFO, the entire SB_{in} FIFO is invalidated, and the missed *sbload* is sent out to memory along with a new set of 15 prefetches. We prefetch the next 15 bursts based on Palacharla and Kessler's work [40] on tuning streaming prefetches. All told, SB_{in} holds 16 entries of 64 bytes each.

Software-defined data streams require a programmer's explicit memory allocation for each table via malloc or similar mechanisms. Because the software side performs all of the memory management, we provide an instruction called `sbpeek` which allows software to peek at the partition ID of the next burst of data to be written from SB_{out} to memory. This allows software to manage the mapping from partition ID to memory address, and to issue the `sbstore` accordingly.

Finally, to ensure seamless execution after an interrupt is encountered, an exception is raised, or a context switch occurs, we make a clean separation of architectural and microarchitectural state. Specifically, only the stream buffers themselves, SB_{in} and SB_{out}, are architecturally visible. No state in the HARP microarchitecture itself is exposed. By design, this separates the microarchitecture of the accelerator from the context, and will help facilitate the use of the stream buffers by other accelerators in the future. Before the machine suspends the HARP execution to service an interrupt or a context switch, the OS will execute an `sbsave` instruction to save all architectural state. After the interrupt has been serviced, before resuming the HARP execution, the OS will execute an `sbrestore` to ensure the machine states are identical before and after the interrupt or context switch. The full stream buffer ISA is summarized in Table 1.

The size of the SB_{out} FIFO is determined by the maximum number of states needed to continue seamless execution. In the baseline instance of the HARP microarchitecture (described further in Section 5.1), we support up to 127 splitters resulting in 256 partitions with 64 bytes per burst. Consequently, the maximum number of in-flight bursts is 256, and so the baseline SB_{out} has 256 entries.

Stream Buffer Instructions
`sbload <address> <record size>`
Load record-sized bytes from specified address in memory to SB_{in}.
`sbstore <address> <record size>`
Store record-sized bytes from SB_{out} to specified address in memory.
`sbpeek <pid>`
Return the *pid* of the burst or records at the head of SB_{out}.
`sbsave`
Save the contents of SB_{in} into memory, drain all in-flight data from HARP to SB_{out}, and save the resulting contents of SB_{out} into memory.
`sbrestore`
Populate SB_{in} and SB_{out} using saved data from memory.

Table 1: Instruction extensions for managing the HARP

HARP Instructions
`set_splitter <splitter number> <value>`
Set the value of a particular splitter (splitter number ranges from 0 to 126).
`partition_start`
Signal HARP to start partitioning and read bursts of records from SB_{in}.
`partition_stop`
Signal HARP to stop partitioning and drain all in-flight data to SB_{out}.

Table 2: Instruction extensions for managing the Hardware-Accelerated Range Partitioner (HARP).

4.2. HARP Instructions

The HARP accelerator itself can be managed via three simple instructions as shown in Table 2. `set_splitter` is invoked once per splitter; `partition_start` signals HARP to start pulling data from the SB_{in}; `partition_stop` signals HARP to stop pulling data from SB_{in} and drain all in-flight data to SB_{out}.

Since the state in the HARP microarchitecture is not visible to other parts of the machine except for the hardware-software streaming framework, the splitter registers do not need to be saved upon interruption. However, all in-flight data in HARP needs to be drained into the SB_{out} before the `sbsave` instruction occurs, in order that HARP resumes execution seamlessly by pulling from the SB_{in} upon `sbrestore` instruction.

These HARP instructions, together with the stream buffer instructions described in the previous section allow full software control of all aspects of the partitioning operation, except for the work of partitioning itself which is handled by HARP.

4.3. The HARP Microarchitecture

We describe a microarchitectural unit, which is tailored to range partition data highly efficiently.

HARP consists of three modules, as depicted in Figure 6. The architecture exploits deep pipelining and simple control

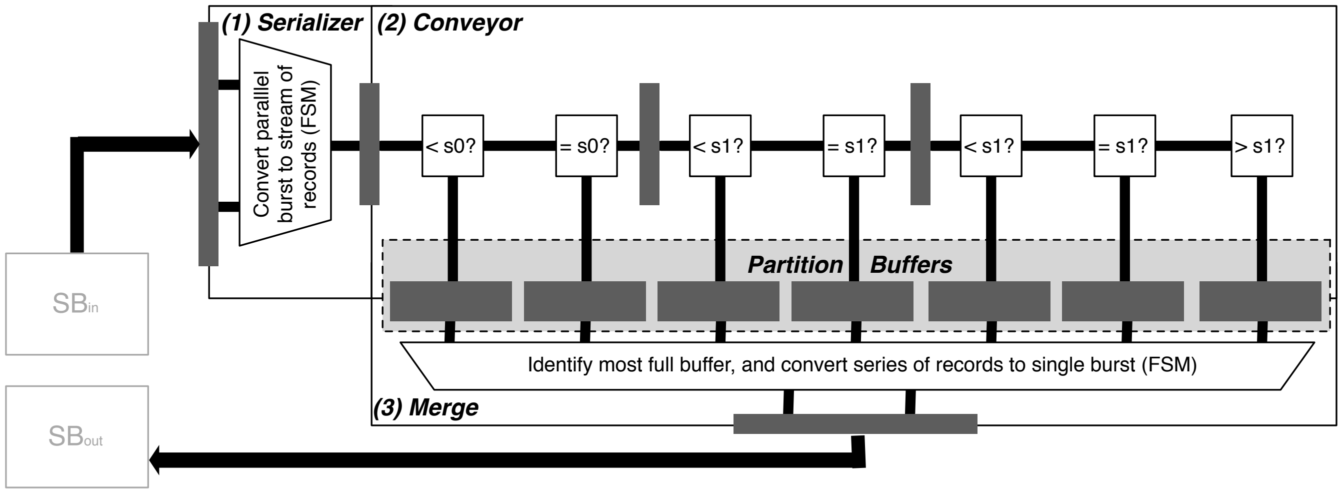
(1) Serializer
(2) Conveyor
Convert parallel burst to stream of records (FSM)
< s0?
= s0?
< s1?
= s1?
< s1?
= s1?
> s1?
Partition Buffers
SBin
SBout
Identify most full buffer, and convert series of records to single burst (FSM)
(3) Merge

Figure 6: HARP draws records in bursts, serializing them into a single stream which is fed into a pipeline of comparators. At each stage of the pipeline, the record key is compared with a splitter value, and the record is either filed in a partition buffer (downwards) or advanced (to the right) according to the outcome of the comparison. As records destined for the same partition collect in the buffers, the merge stage identifies and drains the fullest buffer, emitting a burst of records all destined for the same partition.

flow to keep the pipeline full and maximize throughput. The partitioner pulls and pushes records in 64 byte bursts (tuned to match DRAM).

1. The *serializer* pulls bursts of records in from SB_{in}, and uses a simple finite state machine to pull each individual record from the burst and feed them, one after another, into the subsequent pipeline. As soon as one burst has been fed into the pipe, the serializer is ready to receive the subsequent burst.
2. The *conveyor* module is where the records are compared against splitters. The conveyor accepts a stream of records from the serializer into a deep pipeline with one stage per splitter. At each stage, a record is compared to the corresponding splitter and routed to the “less than” partition, the “equality” partition, or to the next pipeline stage for comparison with the next splitter. When a record is routed to a partition, it is sent to the corresponding partition buffer.
3. As the partition buffers collect records destined for each partition, the *merge* module monitors them, looking for full bursts of records that it can send to a single partition. When a burst is ready, merge drains a burst’s worth of records serially, one record per cycle, bundling them into a single burst and writing it to SB_{out}.

The design uses deep pipelining to hide the latency of multiple splitter comparisons. We experimented with a tree structure for the *conveyor*, analogous to the binary search the software implementation uses, but found that in hardware the linear conveyor architecture was preferable. When the pipeline operates bubble-free, as it does in both cases, it processes one record per cycle, regardless of topology. The only difference in total cycle count between the linear and tree conveyors was the overhead of filling and draining the pipeline at the start and finish respectively. With large record counts, the difference in time required to fill and drain a k-stage pipeline versus a $log(k)$-stage pipe in the tree version, is negligible. While cycle counts were more or less the same between the two, the linear design had a slightly shorter clock period, due to more complex layout and routing requirements in the tree, resulting in slightly better overall throughput.

This HARP unit is record order preserving. All records in a partition appear in the same order they were found in the input record stream. This is a useful property for other parts of the database system and is a natural consequence of the structure of HARP. There is only one route from input port to each partition, and it is impossible for records to pass one another on that route.

5. HARP System Evaluation

First, we evaluate the two principal pieces of the HARP framework, the accelerator itself and the stream buffers, providing a performance comparison with software and detailed cost analysis. Then we evaluate the HARP design flexibility vs. specificity by conducting various sensitivity studies.

5.1. Methodology

Accelerator implementation. We implemented HARP in Bluespec System Verilog [3], a high-level hardware description language that is compiled down to the lower-level language Verilog. Leveraging the parameterizability of Bluespec, we evaluated 11 different points in the partitioner design space.
Accelerator synthesis and physical design. We synthesized each of these designs using the Synopsys [48] Design Compiler with IBM’s 90nm design libraries followed by physical design using the Synopsys IC Compiler. The post-place-and-

route critical path of each design is reported as logic delay plus clock network delay, adhering to the industry standard of reporting critical paths with a margin. We gave the synthesis tools a target clock cycle of less than or equal to 5ns and requested medium effort for area optimization. To compare with the 32 nm Xeon core of the software experiments, HARP area and power numbers are scaled using trend reports from the ITRS roadmap [49]. Using α from ITRS, we apply a shrinking factor of $1/\alpha$ to each dimension of the design and $1/\alpha^2$ for classic scaling of power as described in [46]. The clock frequencies of our designs are scaled more conservatively than area/power following the technology trends reported by the ITRS 90nm to 32nm.

Accelerator simulation. We count hardware cycles using Bluespec's cycle-accurate simulator, Bluesim. Each simulation was performed using 1 million records. Because HARP runs at a lower clock frequency than the system used for the software experiments, we convert cycle counts into absolute bandwidth (in GB/sec).

Baseline accelerator configuration. Each of the analyses in the subsequent sections examines and extends a single baseline HARP configuration. The baseline supports 16 byte records, with 4 byte keys. It has 127 splitters for 255-way partitioning, with 64 byte DRAM bursts (i.e., 4 records per burst). The burst size was selected based on the memory characteristics, rather than the record size.

Software area and power numbers. The per-processor core area and power figures in the analyses that follow reflect our estimates for the system we used. For area, we started with Intel's reported $684mm^2$ die for the E5620 [24]. The eight cores account for roughly half of the die area, so we estimate a single Xeon core to be one sixteenth of the die area or $42.75mm^2$. Similarly, Intel reports a peak thermal design power (TDP) of 80 Watts for each die in this system. Assuming power is roughly proportional to area, we calcuate each core is responsible for roughly one sixteenth of the total, or 5 Watts per core. Strictly speaking TDP is a peak power budget, and actual power consumption is workload dependent. However, because the range partitioning workload is compute-bound and compute-intensive, it is likely running very close to peak power consumption.

5.2. Comparison with Software-Only Partitioning

We compare the range partitioning throughput of the optimistic software from Section 3 with the HARP-augmented version. Figure 7 plots the throughput of three range partitioner implementations: single-threaded software, multi-threaded software, and single HARP. The direct hardware implementation of an algorithm outperforms the serial software equivalent by 7.8X - 9.0X. Here, the throughput difference between hardware and single-threaded software is primarily attributable to the elimination of instruction fetch and control overhead and the deep pipeline. In particular, the structure of the partitioning operation does not introduce hazards or bubbles into the pipeline, allowing it to operate in near-perfect fashion: always full, accepting and emitting one record per clock cycle. This theory is borne out in practice where our empirical measurements indicate average cycles per record ranging from 1.008 (for 15-way partitioning) to 1.041 (for 511-way partitioning). As Figure 7 indicates, it requires 16 threads for the software implementation to match the throughput of the hardware implementation. Moreover, augmenting all 8 cores with a HARP accelerator would provide sufficient compute bandwidth to fully utilize all DRAM pins.

	Stream Buffers				**HARP Unit**			
Num.	Area		Power		Area		Power	
Parts.	mm^2	% Xeon	W	% Xeon	mm^2	% Xeon	W	% Xeon
15	0.07	0.2%	0.063	1.3%	0.08	0.2%	0.005	0.1%
31	0.07	0.2%	0.079	1.6%	0.16	0.4%	0.007	0.1%
63	0.10	0.2%	0.078	1.6%	0.32	0.8%	0.008	0.2%
127	0.11	0.3%	0.085	1.7%	0.66	1.6%	0.015	0.3%
255	**0.13**	**0.3%**	**0.100**	**2.0%**	**1.40**	**3.3%**	**0.029**	**0.6%**
511	0.18	0.4%	0.233	4.7%	2.77	6.5%	0.056	1.1%

Table 3: Area and power overheads of stream buffers and HARP units for various partitioning factors.

The addition of the stream buffer and accelerator hardware do increase the area and power of the core. Table 3 quantifies the area and power overheads of the accelerator and stream buffers relative to a single Xeon core. Comparatively, the additional structures are very small, with the baseline design point adding just 3.6% area and 2.6% power for both the HARP and the SBs. From an energy perspective, the increased power is overwhelmed by the improvement in throughput. Figure 8 compares the partitioning energy per GB of data of software (both serial and parallel) against HARP-based alternatives. The data show a 7.7 - 9.0X improvement in single threaded partitioning energy with HARP. If all eight cores are augmented with HARP, running eight HARP-enhanced cores (with one thread per core) is 5.7 - 6.43X more energy efficient than running sixteen concurrent hyperthreads on those eight cores.

In the following sections we provide a deeper characterization and analysis of the properties of the HARP accelerator, namely its sensitivity to workload, partition fanout, and data sizes.

5.3. Throughput Under Input Skew

We evaluate HARP's sensitivity to skew in the data, by generating synthetically unbalanced record sets, and measuring the partitioner throughput (i.e., cycles/record) on each. We varied the record distribution from optimal, where records were uniformly distributed across all partitions, to pessimal, where all records are sent to a single partition. Figure 9 shows the gentle degradation in throughput as the heavy hitter partition receives an increasingly large share of records.

This degradation is due to the design of the *merge* module. Recall that this stage identifies which partition has the most

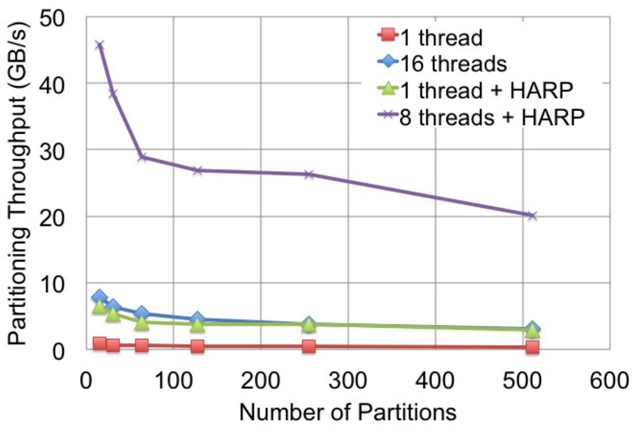

Figure 7: A single HARP unit outperforms single threaded software from 7.8X with 15 partitions to 9.0X with 511 partitions, matching the throughput of 16 threads.

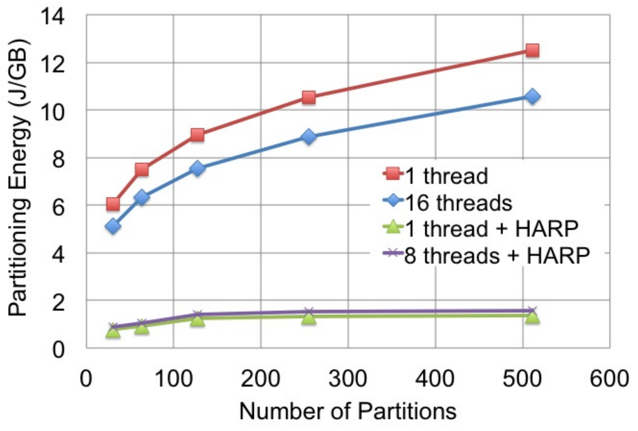

Figure 8: HARP-augmented cores partition data using 5.7-9.0X less energy than parallel or serial software.

records ready and drains them from that partition's buffer to send as a single burst back to memory. When the records are distributed across partitions, the *merge* rarely drains the same partition twice in a row. If B represents the number of records per DRAM burst, draining two different partition buffers back-to-back takes $2B$ cycles. However, when skew increases, the frequency of back-to-back drains of the *same* partition increases. A back-to-back drain of the same partition requires an additional cycle, resulting in an average of $B+1$ cycles per burst rather than B. Thus, the throughput of the *merge* module varies between $\frac{1}{B}$ cycles/record in the best case to $\frac{1}{B+1}$ in the worst case.

The baseline HARP design supports four records per burst resulting in a 25% degradation in throughput between best- and worst-case skew. This is very close to the degradation seen experimentally in Figure 9, where throughput sinks from 3.7 GB/sec with no skew to 3 GB/sec in the worst-case. Note that the worst-case is a function *only* of the number of records per burst, and is bounded at $\frac{B}{B+1}$% of peak. The more records per burst (larger B), the milder the degradation will be. Note that

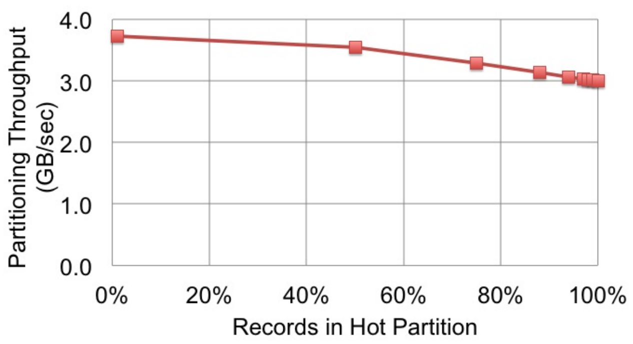

Figure 9: As the input data is increasingly skewed (i.e., unbalanced), the baseline accelerator's throughput drops by up to 19% due to increased occurrence of back-to-back bursts to the same partition.

this tolerance is independent of a number of factors including the number of splitters, the size of the keys, or the size of the table being partitioned.

5.4. Sensitivity to Number of Splitters

Next we analyze how performance, power, and area of HARP change with to the degree of partitioning. Specifically, we vary the number of splitters, $k \in \{7, 15, 31, 63, 127, 255\}$, spanning 15- to 511-way partitioning. We held the key and record widths constant at the baseline values (4 and 16 bytes respectively) and report throughput on a uniform random distribution of records to partitions.

Figure 10 (top) shows the throughput of different HARP sizes, each supporting a different number of splitters, k. While the number of cycles per record held more or less steady between 1.01 and 1.04 cycles per record, the the critical path of the design grows approximately as $k^{0.22}$, causing overall compute bandwidth to decrease in inverse proportion to the critical path.

Both the area and power consumption of the accelerator grow linearly with k, illustrated in Figure 10 (middle) and Figure 10 (bottom) respectively. The number of partition buffers between *conveyor* and *merge*, which account for roughly 50% of the total partitioner area, grows linearly with the number of partitions, which in turn grows linearly with the number of splitters. Meanwhile, the wiring overhead grows slightly super-linearly, due to the increasing number of wires that must be routed. The rest of the combinational logic and other registers and flip-flops scale in a slightly sub-linear fashion. The cumulative result of all these trends is dominated by the partition buffer growth, and thus overall area scales linearly.

Like area, power grows linearly in k. At each design point, the dynamic (or switching) power is a constant 80% of the total power with leakage accounting for the remainder. Under classic technology scaling (i.e., $> 23nm$), power and area are directly proportional, so these two linear trends in Figure 10 (middle) and Figure 10 (bottom) align with expectations.

In terms of absolute numbers, the baseline HARP config-

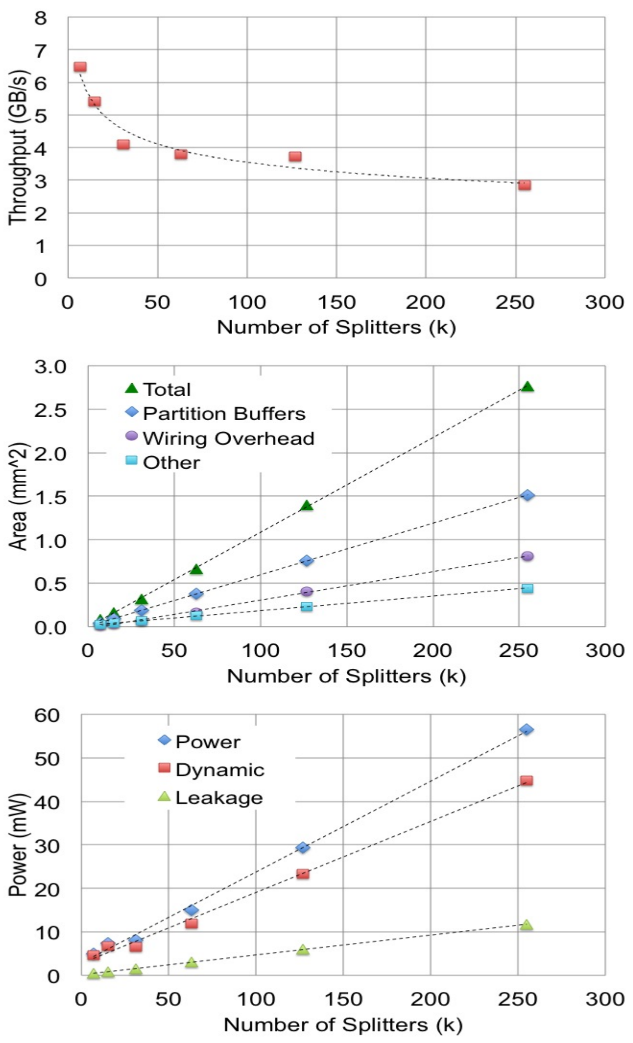

Figure 10: HARP throughput, area, and power for different numbers of splitters, creating a family of small to large HARP instances.

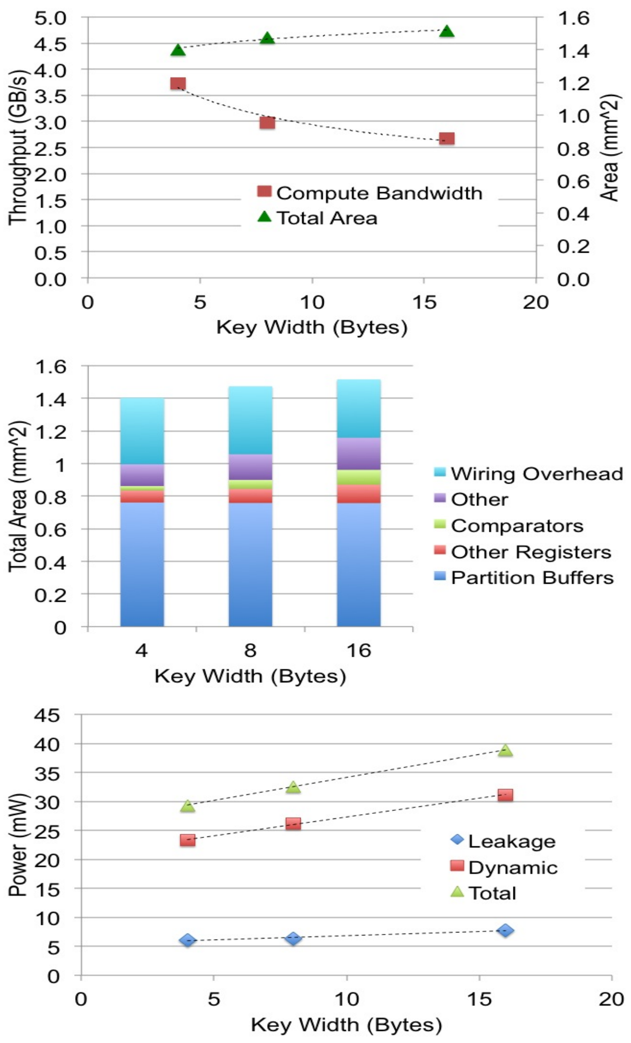

Figure 11: HARP throughput, area, and power at different maximum key widths.

uration ($k = 127$) achieved a 4.26ns critical path, yielding a design that runs at 235 MHz, delivering single-direction compute bandwidth of 3.7 GB/sec. This is 9.3 times faster than single-threaded software range-partitioner described in Section 3. It consumes $1.4mm^2$ and $29.4mW$, or 3.3% and 0.6% of a single Xeon core respectively.

5.5. Sensitivity to Key Width

We now vary the max key width, kw, from the 4 byte baseline to 8 and 16 bytes, the latter of which is the full record width. The critical path increases with the key width, resulting in a compute bandwidth degradation inversely proportional to roughly $(kw)^{0.23}$ as shown on the primary axis of Figure 11 (top). The increasing critical path is due to the comparators in the conveyor module which increase in depth (i.e., levels of gates) as the values being compared grow in width. In the extreme case of 16 byte keys with no payload, the partitioner is able to provide a compute bandwidth of 2.68 GB/sec. Figure 11 (top) also plots area on the secondary axis, showing a logarithmic relation between key width and area. While we see modest area increases, on the whole partitioner area is not especially sensitive to key size.

Figure 11 (middle) shows an area for these three design points, providing several insights. First, the area of the partition buffers remains constant as key width changes. The relative proportion of key to payload has no effect on their overall size. Second, comparator area scales logarithmically with the width of the comparator (i.e. a 32-bit comparator in the $kw = 4$ design will have 5 levels of gates while a 128-bit comparator in the $kw = 16$ design will have 8 levels gates). Third, the area consumed by other registers grows slightly with key width. This area includes both the pipeline registers which do not vary with key width, and the splitter registers which grow linearly with key width. Fourth, as the key width grows, the payload decreases, resulting in reduced overall wiring overhead.

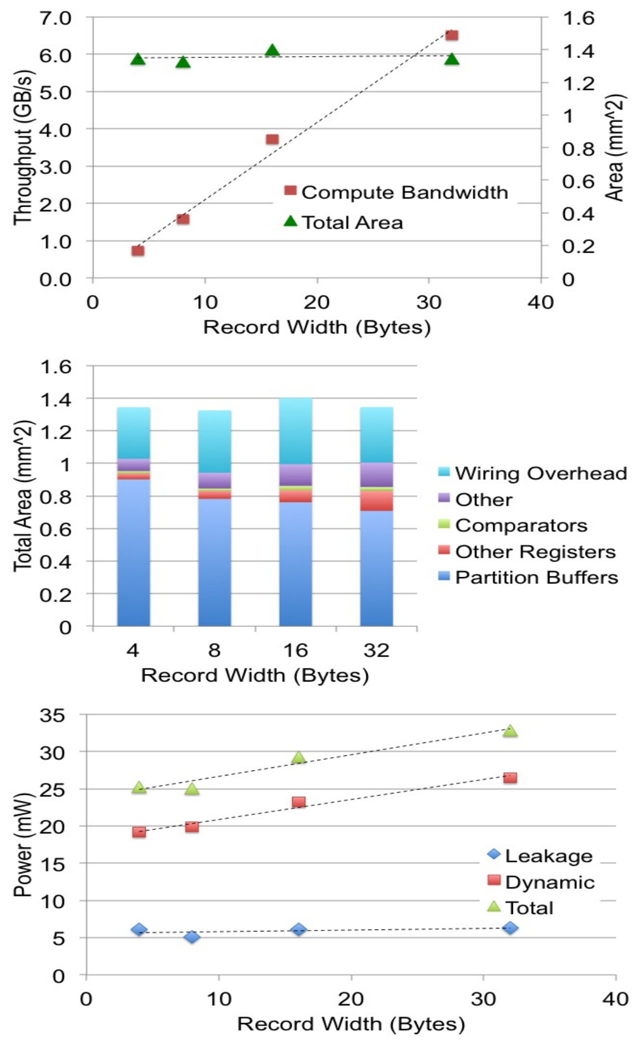

Figure 12: HARP throughput, area, and power at different maximum record widths.

Finally, Figure 11 (bottom) shows how sensitive HARP power is to key width. As before, dynamic power dominates at 80% of total power. Leakage power increases only slightly while dynamic power increases more rapidly but not nearly as sensitive to key width as opposed to number of splitters.

5.6. Sensitivity to Record Width

As we found in the key width sensitivity analysis, many aspects of the HARP design were more sensitive to record width than key width. So, we now vary the record width, *rw*, for $rw \in \{4, 8, 16, 32\}$, while keeping all other aspects of the baseline configuration.

We first observe, in Figure 12 (top), that the compute bandwidth of the partitioner doubles as the record width doubles. The reason is that the rate of which records are processed held steady (at about 1.01 cycles per record) and the record width doubled, doubling the total bandwidth. While the critical paths of the designs had a slight effect on bandwidth, the record width trend dominated. In contrast, partitioner area stayed mostly constant as *rw* increased.

Breaking down the total area into five categories as shown in Figure 12 (middle), we make the following two observations. First, partition buffers got smaller as *rw* got bigger. As burst size was constant throughout, this result shows that the depth of the buffer costs more in area than the width of the buffer, i.e., an 8-entry, 16-bit wide queue requires more space than a 4-entry, 32-bit wide queue. This is a result of the per-entry state required to maintain the queue. Second, as we saw before, comparator area is a function of key width, and thus it is now more or less constant as record size changes. Finally, Figure 12 (bottom) shows similar power trends to key width: power grows linearly, with dynamic power accounting for 80% of the total, and leakage power is proportional to area (here, roughly constant).

In closing, we summarize these sensitivity studies as follows. HARP's partitioning throughput is most sensitive to record width growing linearly in direct proportion, while area and power are most sensitive to the number of splitters, also growing linearly in direct proportion.

6. Related Work

6.1. Streaming Computation

The last decade has seen substantial interest in software-based streaming computation with the development of new parallel languages [6, 16, 7] and middleware support focused on portability and interoperability [10, 26, 38, 12, 11].

The hardware support for streaming has been substantially more limited. The vast majority of streaming architectures, such as Cell's SPE [14], RSVP [9], or Piperench [15] are decoupled from the processing core and are highly tailored to media processing. The designs that most closely resemble HARP microarchitecturally are DySER [18] and ReMAP [50]. DySER incorporates a dynamically specializable datapath into the core. Both DySER and HARP can be viewed as specialized functional units, and are sized accordingly (a couple percent of a core area). While one might be able to program DySER to partition data, its full interconnect between functional units is overkill for partitioning's predicatble data flow. ReMAP [50] has a very different goal, integrating reconfigurable fabric, called Specialized Programmable Logic (or SPL), to support fine-grained inter-core communication and computation.

6.2. Vector ISAs

Nearly all modern processors include vector ISAs, exemplified by x86's MMX and SSE, Visual Instruction Set (VIS) for UltraSPARC, or AltiVec on PowerPC. These ISAs include vector loads and stores, instructions which load 128- or 256-bit datawords into registers for SIMD vector operation. Different opcodes allow the programmer to specify whether the data should or should not be cached (e.g., non-temporal loads).

The SIMD vector extensions outlined above were universally introduced to target media applications on streaming video and image data. The available operations treat the data as vectors and focus largely on arithmetic and shuffling operations on the vector values. Many programmers have retrofitted and vectorized other types of programs, notably text parsing [5, 31] and regular expression matching [44] and database kernels [52, 17, 30]. Our experiments in Section 3 using a state of the art SIMD range partitioning [45] indicate that vector-based traversal improves throughput but fails to fully saturate DRAM bandwidth.

These efforts demonstrate moderate speedups, at the cost of substantial programmer effort. One recent study of regular expression matching compared different strategies for acceleration [44]. The study concluded that SIMD software was the best option, due to the fast data and control transfers between the scalar CPU and the vector unit. The other approaches (including memory bus and network attached accelerators) suffered due to communication overheads. In short, SIMD won not because it was particular fast computationally, but because it was fast to invoke. This study in part influenced our choice to couple the HARP accelerator with a processing core.

6.3. Database Machines

Database machines were developed by the database community in the early 1980s as specialized hardware for database workloads. These efforts largely failed, primarily because commodity CPUs were improving so rapidly at the time, and hardware design was slow and expensive [4]. While hardware design remains quite costly, high computing requirements of data-intensive workloads, limited single-threaded performance gains, increases in specialized hardware, aggressive efficiency targets, and the data deluge have spurred us and others to revisit this approach. While FPGAs have been successfully used to accelerate a number of data intensive algorithms [34, 51, 35], they are power-hungry compared to custom logic and it remains unclear how to approach programming and integrating them.

6.4. Memory Scheduling

Despite the relative scarcity of memory bandwidth, there is ample evidence both in this paper and elsewhere that workloads do not fully utilize the available resource. One recent study suggests that data bus utilization would double, LLC miss penalties would halve, and overall performance would increase by 75% if memory controllers were to operate at their peak throughput [25]. This observation and others about the performance criticality of memory controller throughput [37] have inspired substantial research in memory scheduling [20, 32, 21, 42, 53, 41, 47, 25, 13, 36, 29]. Improvements in memory controllers have the advantage of being applicable across all workloads, but important throughput bound workloads, such as partitioning, are not limited by the memory controller and thus will not see significant benefit from those efforts.

7. Conclusions

We proposed a database processing streaming framework and a database processing element accelerator architecture that provide seamless execution in modern computer systems and exceptional throughput and power efficiency advantages. These benefits are necessary to address the ever increasing demands of big data processing. This proposed framwork can be utilized for other database processing accelerators such as specialized aggregators, joiners, sorters, and so on, setting forth a flexible yet modular data-centric acceleration framework.

We presented the design and implementation of HARP, a hardware accelerated range partitioner. HARP is able to provide a compute bandwidth of at least 9.3 times a very efficient software algorithm running on a state-of-the-art Xeon core, with just 3.6% of the area and 2.6% of the power. Processing data with accelerators such as HARP can alleviate serial performance bottlenecks in the application and can free up resources on the server to do other useful work, getting us one step closer to closing the dark silicon gap in designing a more efficient computer system.

References

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, "DBMSs on a modern processor: Where does time go?" in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1999.

[2] S. Blanas, Y. Li, and J. M. Patel, "Design and evaluation of main memory hash join algorithms for multi-core CPUs," in *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2011.

[3] Bluespec, Inc., "Bluespec Core Technology." [Online]. Available: http://www.bluespec.com

[4] H. Boral and D. J. DeWitt, "Database machines: an idea whose time has passed?" in *Proceedings of the Second International Workshop on Database Machines*, 1983.

[5] R. D. Cameron and D. Lin, "Architectural support for SWAR text processing with parallel bit streams: the inductive doubling principle," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[6] S. Chakraborty and L. Thiele, "A new task model for streaming applications and its schedulability analysis," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2005.

[7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Proceedings of the Annual Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, 2005.

[8] J. Cieslewicz and K. A. Ross, "Data partitioning on chip multiprocessors," in *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, 2008.

[9] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi, "The reconfigurable streaming vector processor (RSVPTM)," in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 2003.

[10] B. F. Cooper and K. Schwan, "Distributed stream management using utility-driven self-adaptive middleware," in *Proceedings of the International Conference on Automatic Computing*, 2005.

[11] M. Duller and G. Alonso, "A lightweight and extensible platform for processing personal information at global scale," *Journal of Internet Services and Applications*, vol. 1, no. 3, pp. 165–181, Dec. 2010.

[12] M. Duller, J. S. Rellermeyer, G. Alonso, and N. Tatbul, "Virtualizing stream processing," in *Proceedings of International Conference on Middleware*, 2011.

[13] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt, "Parallel application memory scheduling," in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 2011.

[14] B. Flachs, S. Asano, S. Dhong, P. Hotstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H. Oh, S. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, and N. Yano, "A streaming processing unit for a CELL processor," in *Proceedings of the International Solid-State Circuits Conference (ISSCC)*, 2005.

[15] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer, "PipeRench: a co/processor for streaming multimedia acceleration," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1999.

[16] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[17] N. K. Govindaraju and D. Manocha, "Efficient relational database management using graphics processors," in *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, 2005.

[18] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *Proceedings of the Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[19] G. Graefe and P.-A. Larson, "B-tree indexes and CPU caches," in *Proceedings of the International Conference on Data Engineering*, 2001.

[20] S. I. Hong, S. A. McKee, M. H. Salinas, R. H. Klenke, J. H. Aylor, and W. A. Wulf, "Access order and effective bandwidth for streams on a direct Rambus memory," in *Proceedings of the Symposium on High Performance Computer Architecture (HPCA)*, 1999.

[21] I. Hur and C. Lin, "Adaptive history-based memory schedulers," in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 2004.

[22] IBM, "DB2 Partitioning Features." [Online]. Available: http://www.ibm.com/developerworks/data/library/techarticle/dm-0608mcinerney/

[23] ——, "IBM What is big data? Bringing big data to enterprise." [Online]. Available: http://www-01.ibm.com/software/data/bigdata/

[24] Intel Corporation, "Intel® Xeon® Processor E5620." [Online]. Available: http://ark.intel.com/products/47925/Intel-Xeon-Processor-E5620-(12M-Cache-2_40-GHz-5_86-GTs-Intel-QPI)

[25] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2008.

[26] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani, "Design, implementation, and evaluation of the linear road bnchmark on the stream processing core," in *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2006.

[27] N. P. Jouppi, "Improvind direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1990.

[28] C. Kim, E. Sedlar, J. Chhugani, T. Kaldewey, A. D. Nguyen, A. D. Blas, V. W. Lee, N. Satish, and P. Dubey, "Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 2, no. 2, pp. 1378–1389, 2009.

[29] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 2010.

[30] J. Krueger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier, "Fast updates on read-optimized databases using multi-core CPUs," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 5, no. 1, pp. 61–72, Sep. 2011.

[31] D. Lin, N. Medforth, K. S. Herdy, A. Shriraman, and R. Cameron, "Parabix: Boosting the efficiency of text processing on commodity processors," in *Proceedings of the Symposium on High Performance Computer Architecture (HPCA)*, 2012.

[32] S. A. McKee, W. A. Wulf, J. H. Aylor, M. H. Salinas, R. H. Klenke, S. I. Hong, and D. A. B. Weikle, "Dynamic access ordering for streamed computations," *IEEE Transactions on Computers*, vol. 49, no. 11, pp. 1255–1271, Nov. 2000.

[33] Microsoft, "Microsoft SQL Server 2012." [Online]. Available: http://technet.microsoft.com/en-us/sqlserver/ff898410

[34] C. Mohan, "Impact of recent hardware and software trends on high performance transaction processing and analytics," in *Proceedings of the TPC Technology Conference on Performance Evaluation, Measurement and Characterization of Complex Systems (TPCTC)*, 2011.

[35] R. Müller and J. Teubner, "FPGAs: a new point in the database design space," in *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2010.

[36] S. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing memory interference in multicore systems via application-aware memory channel partitioning," in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 2011.

[37] C. Natarajan, B. Christenson, and F. Briggs, "A study of performance impact of memory controller features in multi-processor server environment," in *Proceedings of Workshop on Memory Performance Issues (WMPI)*, 2004.

[38] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *International Conference on Data Mining Workshops (ICDMW)*, 2010.

[39] Oracle, "Oracle Database 11g: Partitioning." [Online]. Available: http://www.oracle.com/technetwork/database/options/partitioning/index.html

[40] S. Palacharla and R. E. Kessler, "Evaluating stream buffers as a secondary cache replacement," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1994.

[41] N. Rafique, W.-T. Lim, and M. Thottethodi, "Effective Management of DRAM Bandwidth in Multicore Processors," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2007.

[42] S. Rixner, "Memory controller optimizations for web servers," in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 2004.

[43] K. A. Ross and J. Cieslewicz, "Optimal splitters for database partitioning with size bounds," in *Proceedings of the International Conference on Database Theory (ICDT)*, 2009, pp. 98–110.

[44] V. Salapura, T. Karkhanis, P. Nagpurkar, and J. Moreira, "Accelerating business analytics applications," in *Proceedings of the Symposium on High Performance Computer Architecture (HPCA)*, 2012.

[45] B. Schlegel, R. Gemulla, and W. Lehner, "k-ary search on modern processors," in *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, 2009.

[46] O. Shacham, O. Azizi, M. Wachs, W. Qadeer, Z. Asgar, K. Kelley, J. P. Stevenson, A. Solomatnikov, A. Firoozshahian, B. Lee, S. Richardson, and M. Horowitz, "Rethinking digital design: Why design must change," *IEEE Micro*, vol. 30, no. 6, pp. 9–24, NovDec 2010.

[47] J. Shao and B. Davis, "A burst scheduling access reordering mechanism," 2007.

[48] Synopsys, Inc., "90nm Generic Library for IC Design, Design Compiler, IC Compiler." [Online]. Available: http://www.synopsys.com

[49] The International Techonology Roadmap for Semiconductors, "ITRS executive summary," Tech. Rep., 2009. [Online]. Available: http://www.itrs.com

[50] M. A. Watkins and D. H. Albonesi, "ReMAP: A reconfigurable heterogeneous multicore architecture," in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 2010.

[51] L. Woods, J. Teubner, and G. Alonso, "Complex event detection at wire speed with FPGAs," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 3, no. 1, pp. 660–669, 2010.

[52] J. Zhou and K. A. Ross, "Implementing database operations using SIMD instructions," in *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2002.

[53] Z. Zhu and Z. Zhang, "A performance comparison of DRAM memory system optimizations for smt processors," in *Proceedings of the Symposium on High Performance Computer Architecture (HPCA)*, 2005.